Recursive SQL for Data Mining

Maximilian E. Schüle
Technical University of Munich
m.schuele@tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

ABSTRACT
To implement algorithms within database systems beyond the design of SQL as a data query language, library functions or external tools were used that require the extraction of data first. To eliminate the need of data extraction out of database systems, we argue that SQL-92 plus recursive tables is capable of expressing user-defined algorithms. To underline this claim, we transform selected algorithms out of graph mining, clustering and association rule analysis into recursive common table expressions (CTEs). We compare their performance to the one of user-defined functions and external tools. Our evaluation shows a competitive performance when using recursive CTEs to library functions either when using a disk-based database systems or a modern in-memory engine.

ACM Reference Format:

1 INTRODUCTION
The performance increase of database servers through modern hardware allows database systems to be used for more than pure data management tasks [2, 6, 9, 15, 19, 38, 40, 41]. Database systems provide with SQL a declarative language to specify what to do rather than caring about optimisation details. The platform independence and reusability of SQL increases the incentive to execute complex algorithms already in the database system [3, 7, 8, 17]. SQL provides common table expressions (CTEs) that allow structuring and modularising an SQL query and which are part of the holistic query optimisation [6]. Furthermore, recursive CTEs compute the transitive closure, which allow SQL-92 to be Turing-complete.

In order for database servers to take over more application logic, database users should be able to develop algorithms independently of systems developers. In a previous publication [34], we argued that a domain-specific language with procedural constructs and embedded SQL (HyPerScript) is needed to eliminate the need for developing an operator in each case. So we identified building blocks to implement selected data mining algorithms as user-defined functions (UDFs) [5, 20, 39, 42] in SQL.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SSDBM 2022, July 6–8, 2022, Copenhagen, Denmark.
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9667-7/22/07...$15.00
https://doi.org/10.1145/3538712.3538746

In this study, we argue that transferring procedural constructs into SQL-92 is possible. We start by converting each iteration within a procedural loop into a recursion [12, 13, 16] to provide SQL-92 only implementations for clustering (DBSCAN, k-Means), graph mining (PageRank) and association rule analysis (Apriori). We compare their performance in PostgreSQL and HyPer [24, 25], an in-memory database system [21, 22, 27, 32, 35], to their UDF counterparts and dedicated table-operators: MADlib [10] in PostgreSQL and data mining operators in HyPer [11, 18, 31, 37].

This study’s contributions are data mining operators written in SQL-92 only and a comprehensive evaluation to compare their performance to existing library functions and procedural counterparts as part of UDFs. This paper is structured as follows: Section 2 describes the implementation of data mining algorithms in SQL. The evaluation in Section 3 measures the performance in dependency on the input size and the number of available threads. Section 4 concludes by an outlook on further applications for recursive CTEs.

2 SQL FOR DATA ANALYSIS
This section presents a set of data analysis algorithms written in recursive SQL algorithms for association rule analysis (Apriori), clustering (k-Means and DBSCAN) and graph metrics (PageRank).

2.1 PageRank
PageRank [4] is a graph mining algorithm designed to determine the importance of web pages. Each web page is called a node \( n \in N \); a link directing to another web page is called an edge \( (s, d) \in E \). Initially, each node receives the same PageRank value \( p_r(n) = \frac{1}{|N|} \) (Equation 1). In each iteration, each node \( s \) distributes its own value \( p_r(s) \) equally to all outgoing edges \( (s, d) \in E \). The new PageRank value \( p_r(s) \) of a node \( s \) is the sum of the values of all incoming edges \( (s, d) \in E \), possibly damped by a factor \( \alpha \) (Equation 2):

\[
p_r(n) := \frac{1}{|N|}. \quad (1)
\]

\[
p_r(s) = \sum_{(s, d) \in E} \frac{p_r(s)}{|d|} + \frac{1 - \alpha}{|N|}. \quad (2)
\]

Listing 1: PageRank in SQL with a recursive table pagerank

```sql
WITH recursive pagerank (iter, node, pr) AS (  
  SELECT 1, e.dst, 1.0/float((SELECT count(DISTINCT dst) FROM prscript.edges))  
  UNION ALL  
  SELECT iter + 1, e.dst, p.r/((1.0/float(SELECT count(DISTINCT dst) FROM prscript.edges)) + 0.8 * sum(b))  
  FROM (  
    SELECT iter, e.dst, p.r * (SELECT count(*) FROM prscript.edges WHERE x.src = e.src) AS b  
    FROM prscript.edges e, pagerank p  
    WHERE e.src = p.node AND iter < 100  
  ) AS s, x  
  WHERE x.src = s.dst AND iter = 100  
)  
SELECT * FROM pagerank WHERE iter = 100;
```
Both equations can be described in SQL with aggregations, whereas for the iteration we need either loops of a scripting language like HyperScript or a recursive table (see Listing 1). We expect one relation containing the edges. The base case (line 2/3) computes the initial PageRank value pr₀. The recursive step first divides each node’s PageRank value by the number of outgoing edges (line 7) and assigns this fraction to the destination node dst. It then sums up the fractions for each destination node (line 5-10).

2.2 Clustering
Clustering is an important area of data analysis that groups similar tuples. We consider k-Means and DBSCAN [23], as both algorithms are integrated into Hyper as operators.

k-Means assigns n-dimensional points \( x \in P \subset \mathbb{R}^n \) to \( k \) clusters \( C \) with \( k \) points forming the initial centres \( C_0 \subset P, |C_0| = k \). A point belongs to the closest located cluster \( c \in C \) based on a metric like Euclidean distance \( ||c - x||_2 \), Equation 3 returns all points of a cluster. In each iteration, the new centre is computed as the average of all points (Equation 4):

\[
\text{cluster}(c) = \{ x \in P : \exists d \in C : d \neq c \land ||d - x||_2 < ||c - x||_2 \}.
\]

\[
c_{i+1} = \sum_{x \in \text{cluster}(c_i)} \frac{x}{|\text{cluster}(c_i)|}.
\]

Using a recursive table (see Listing 2), \( k \) tuples are initially selected as centres (line 2), whose coordinates get updated in each iteration (line 4-8). The implementation for computing the centres of k-Means is based on a window function that calculates a ranking of the closest centres per point (line 5/6). The mean values of the assigned points form the new centres (line 4).

For comparison, we use the operators k-Means and DBSCAN implemented in Hyper. As a special feature, both operators are written directly in LLVM code, are compiled directly into the query and avoid expensive function calls. Both operators are pipeline breakers, but the centre calculation for k-Means does not require materialisation of the data points as it copies the underlying operator tree per iteration.

2.3 Association Rule Analysis
The Apriori algorithm [1], introduced in 1993, is the best-known representative in the field of association rule analysis. It is based on shopping cart data stored as tuples out of transaction number \( (tid) \) and item (sales: \{ \( \{tid, item\}\}\}). First, it selects frequent item sets that occur with a minimum relative frequency, support, of at least \( x_0 \) in all shopping carts, which are used to create association rules. In each iteration, the item sets grow by one element, starting with the one-element set. Here, the number of iterations and item sets to be checked is limited by the apriori principle. The principle states that an item set whose subsets do not occur frequently cannot be a frequent item set.

The recursive implementation (see Listing 4) is based on an array representation for frequent item sets, which are iteratively extended with recursive SQL. Here, arrays are used as sets and expanded by one element in each iteration. Then the support of each item set is counted. For this purpose, each item set is compared with each shopping cart using the set operator \( \&\& \) shopping cart (line 13).

DBSCAN clusters points depending on a parameter \( \epsilon \), that describes the maximal distance between two points, and the minimal number of points per cluster \( minPoints > 1 \), declared as noise otherwise. For every point within a cluster \( x \in G \subset P \subset \mathbb{R}^n \), another point \( y \in G \) exists, whose distance to \( x \) is less than \( \epsilon \):

\[
\forall x \in G : \exists y \in G : x \neq y \land ||x - y||_2 < \epsilon.
\]

When the database system supports aggregate functions within recursive tables, the clusters can be expanded recursively (see Listing 3): First, each point forms its own cluster (line 2). Then, clusters that are less than \( \epsilon \) away are merged (line 4-7).

Listing 2: k-Means in SQL with a recursive table clusters: \( k=5 \), 100 iterations.

DBSCAN clusters points depending on a parameter \( \epsilon \), that describes the maximal distance between two points, and the minimal number of points per cluster \( minPoints > 1 \), declared as noise otherwise. For every point within a cluster \( x \in G \subset P \subset \mathbb{R}^n \), another point \( y \in G \) exists, whose distance to \( x \) is less than \( \epsilon \):

\[
\forall x \in G : \exists y \in G : x \neq y \land ||x - y||_2 < \epsilon.
\]

Listing 3: DBSCAN in SQL with a recursive table dbscan: 10 iterations, \( \epsilon = 1.5 \), \( minPoints = 3 \).

Listing 4: Determining the frequent item sets for the Apriori algorithm: a recursively growing relation calculates the frequent item sets, starting with the one-element item sets (each item as an array with one element).

The operator implemented in Hyper is based on storing the elements in a prefix tree that grows with each iteration. Special features of the implementation in Hyper are the parallelism per iteration step.
and the handling of duplicates. Thus, the association rules consider the support of identical elements within a shopping cart.

3 EVALUATION

The evaluation compares the performance of procedures created with HyPerScript to table operators of HyPer and MADlib, recursive tables in HyPer, PostgreSQL and Umbra, and PL/pgSQL procedures (PostgreSQL 12.6 with MADlib 1.17.0 extension). All experiments were measured on a Ubuntu 20.04 LTS machine with six Intel Core i7-3930K CPUs running at 3.20 GHz and 64 GB DDR4 RAM.

For the Apriori algorithm, 100 different items and 1000 shopping carts were synthetically generated. The number of items per shopping cart varied between 0 and 10. For clustering, we generated $10^6$ points whose x- and y-coordinates were equally distributed in the interval $[0, 10^6]$. The PageRank value was computed for $10^5$ nodes with the same number of edges. All experiments were repeated three times and the median was taken for the measurements.

For the Apriori algorithm, we varied the minimum support (the larger, the less frequent item sets exist, see Figure 1). With increasing minimum support, the runtime decreases as less frequent item sets exist. Although the HyPer operator performs the best, the implementation in HyPerScript is slower than its counterpart in PL/pgSQL. This is caused by an implementation of the array set operator within HyPer that unnests the array internally.

The runtimes of the clustering algorithms grow linearly with the input size (see Figure 2a). Although the integrated operators perform the best, the k-Means implementation in HyPerScript is slower than its counterpart in PostgreSQL. The recursive computation of DBSCAN in Umbra [14, 29, 30] (as the other database systems do not support min as aggregate function inside a recursive table) was as fast as the implemented operator. The k-Means algorithm in HyPerScript computes 30 % faster with each additional core (see Figure 2b), as the underlying database system executes the SQL queries in parallel, whereas the k-Means operator is explicitly parallellised. Neither DBSCAN as an operator nor as a stored procedure support scaling. The SQL queries used in the HyPerScript procedure scale poorly because only one tuple is initialised as a new cluster per (non-parallelised) iteration.

All implementations of PageRank in HyPer outperform their counterparts in PostgreSQL (see Figure 4). The implementations...
within a scripting language perform slightly worse than the corresponding query using a recursive table in PostgreSQL and Hyper respectively. With few edges, the additional overhead of the integrated operator in Hyper becomes apparent, since it creates a dictionary for the nodes and stores edges in a sparse matrix as Compressed-Sparse-Row (CSR). With an increasing number of edges, the additional effort for the dictionary and the CSR data structure is amortised, so that the operator calculates the PageRank value faster than the script function.

4 CONCLUSION

This study showed how to express data mining algorithms in SQL-92 only by relying on recursive CTEs. For this reason, we implemented four algorithms, namely k-Means, DBSCAN, Apriori and PageRank in SQL and compared their performance to library functions and UDFs. The evaluation revealed that the recursive implementation performs worse than the operators in Hyper but due to the performance of an in-memory database system similar to MADlib’s library functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of functions.

REFERENCES